Liquidsoap: a High-Level Programming Language for Multimedia Streaming

David Baelde, Romain Beauxis, Samuel Mimram

▶ **To cite this version:**

David Baelde, Romain Beauxis, Samuel Mimram. Liquidsoap: a High-Level Programming Language for Multimedia Streaming. SOFSEM 2011: Theory and Practice of Computer Science, Jan 2011, Nový Smokovec, Slovakia. pp.99-110, 10.1007/978-3-642-18381-2_8 . inria-00585728

HAL Id: inria-00585728
https://inria.hal.science/inria-00585728

Submitted on 13 Apr 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Liquidsoap: a High-Level Programming Language for Multimedia Streaming

David Baelde[1], Romain Beauxis[2], and Samuel Mimram[3]

[1] University of Minnesota, USA
[2] Department of Mathematics, Tulane University, USA
[3] CEA LIST – LMeASI, France

Abstract. Generating multimedia streams, such as in a netradio, is a task which is complex and difficult to adapt to every users' needs. We introduce a novel approach in order to achieve it, based on a dedicated high-level functional programming language, called *Liquidsoap*, for generating, manipulating and broadcasting multimedia streams. Unlike traditional approaches, which are based on configuration files or static graphical interfaces, it also allows the user to build complex and highly customized systems. This language is based on a model for streams and contains operators and constructions, which make it adapted to the generation of streams. The interpreter of the language also ensures many properties concerning the good execution of the stream generation.

The widespread adoption of broadband internet in the last decades has changed a lot our way of producing and consuming information. Classical devices from the analog era, such as television or radio broadcasting devices have been rapidly adapted to the digital world in order to benefit from the new technologies available. While analog devices were mostly based on hardware implementations, their digital counterparts often consist in software implementations, which potentially offers much more flexibility and modularity in their design. However, there is still much progress to be done to unleash this potential in many areas where software implementations remain pretty much as hard-wired as their digital counterparts.

The design of domain specific languages is a powerful way of addressing that challenge. It consists in identifying (or designing) relevant domain-specific abstractions (construct well-behaved objects equipped with enough operations) and make them available through a programming language. The possibility to manipulate rich high-level abstractions by means of a flexible language can often release creativity in unexpected ways. To achieve this, a domain-specific language should follow three fundamental principles. It should be

1. *adapted*: users should be able to perform the required tasks in the domain of application of the language;
2. *simple*: users should be able to perform the tasks in a simple way (this means that the language should be reasonably concise, but also understandable by users who might not be programming language experts);

3. *safe*: the language should perform automatic checks to prevent as many errors as possible, using static analysis when possible.

Balancing those requirements can be very difficult. This is perhaps the reason why domain specific languages are not seen more often. Another reason is that advanced concepts from the theory of programming language and type systems are often required to obtain a satisfying design.

In this paper, we are specifically interested in the generation of multimedia streams, notably containing audio and video. Our primary targets are netradios, which continuously broadcast audio streams to listeners over Internet. At first, generating such a stream might seem to simply consist in concatenating audio files. In practice, the needs of radio makers are much higher than this. For instance, a radio stream will often contain commercial or informative jingles, which may be scheduled at regular intervals, sometimes in between songs and sometimes mixed on top of them. Also, a radio program may be composed of various automatic playlists depending on the time of the day. Many radios also have live shows, based on a pre-established schedule or not; a good radio software is also expected to interrupt a live show when it becomes silent. Most radios want to control and process the data before broadcasting it to the public, performing tasks like volume normalization, compression, etc. Those examples, among many others, show the need for very flexible and modular solutions for creating and broadcasting multimedia data. Most of the currently available tools to broadcast multimedia data over the Internet (such as Darkice, Ezstream, VideoLAN, Rivendell or SAM Broadcaster) consist of straightforward adaptation of classical streaming technologies, based on predefined interfaces, such as a virtual mixing console or static file-based setups. Those tools perform very well a predefined task, but offer little flexibility and are hard to adapt to new situations.

In this paper, we present Liquidsoap, a domain-specific language for multimedia streaming. Liquidsoap has established itself as one of the major tools for audio stream generation. The language approach has proved successful: beyond the obvious goal of allowing the flexible combination of common features, unsuspected possibilities have often been revealed through clever scripts. Finally, the modular design of Liquidsoap has helped its development and maintenance, enabling the introduction of several exclusive features over time. Liquidsoap has been developed since 2004 as part of the Savonet project [2]. It is implemented in OCaml, and we have thus also worked on interfacing many C libraries for OCaml. The code contains approximatively 20K lines of OCaml code and 10K lines of C code and runs on all major operating systems. The software along with its documentation is freely available [2] under an open-source license.

Instead of concentrating on detailing fully the abstractions manipulated in Liquidsoap (streams and sources) or formally presenting the language and its type system, this paper provides an overview of the two, focusing on some key aspects of their integration. We first give a broad overview of the language and its underlying model in Section 1. We then describe two recent extensions of that basic setup. In Section 2 we illustrate how various type system features are combined to control the combination of stream of various content types. Section 3

motivates the interest of having multiple time flows (clocks) in a streaming system, and presents how this feature is integrated in Liquidsoap. We finally discuss related systems in Section 4 before concluding in Section 5.

1 Liquidsoap

1.1 Streaming model

A stream can be understood as a timed sequence of data. In digital signal processing, it will simply be an infinite sequence of samples – floating point values for audio, images for video. However, multimedia streaming also involves more high-level notions. A *stream*, in Liquidsoap, is a succession of *tracks*, annotated with *metadata*. Tracks may be finite or infinite, and can be thought of as individual songs on a musical radio show. Metadata packets are punctual and can occur at any instant in the stream. They are used to store various information about the stream, such as the title or artist of the current track, how loud the track should be played, or any other custom information. Finally, tracks contain multimedia data (audio, video or MIDI), which we discuss in Section 2.

Streams are generated on the fly and interactively by *sources*. The behavior of sources may be affected by various parameters, internal (*e.g.*, metadata) or external (*e.g.*, execution of commands made available via a server). Some sources purely produce a stream, getting it from an external device (such as a file, a sound card or network) or are synthesizing it. Many other sources are actually operating on other sources in the sense that they produce a stream based on input streams given by other sources. Abstractly, the program describing the generation of a stream can thus be represented by a directed acyclic graph, whose nodes are the sources and whose edges indicate dependencies between sources (an example is given in Figure 1).

Some sources have a particular status: not only do they compute a stream like any other source, but they also perform some observable tasks, typically outputting their stream somewhere. These are called *active* sources. Stream generation is performed "on demand": active sources actively attempt to produce their stream, obtaining data from their input sources which in turn obtain data from their dependent sources, and so on. An important consequence of this is the fact that *sources do not constantly stream*: if a source would produce a stream which is not needed by any active source then it is actually frozen in time. This avoids useless computations, but is also crucial to obtain the expected expressiveness. For example, a `rotation` operator will play alternatively several sources, but should only rotate at the end of tracks, and its unused sources should not keep streaming, otherwise we might find them in the middle of a track when we come back to playing them. *Sources are also allowed to fail* after the end of a track, *i.e.*, refuse to stream, momentarily or not. This is needed, for example, for a queue of user requests which might often be empty, or a playlist which may take too long to prepare a file for streaming. Failure is handled by various operators, the most common being the *fallback*, which takes a list of sources and replays the stream of the first available source, failing when all of them failed.

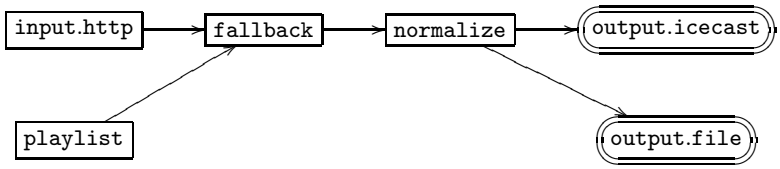

Fig. 1. A streaming system with sharing

1.2 A language for building streaming systems

Based on the streaming model presented above, Liquidsoap provides the user with a convenient high-level language for describing streaming systems. Although our language borrows from other functional programming languages, it is has been designed from scratch in order to be able to have a dedicated static typing discipline together a very user-friendly language.

One of the main goals which has motivated the design of the Liquidsoap language is that it should be very accessible, even to non-programmers. It turned out that having a functional programming language is very natural (cf. Section 1.3). The built-in functions of the language often have a large number of parameters, many of which have reasonable default values, and it would be very cumbersome to have to write them all each time, in the right order. In order to address this, we have designed a new extension of λ-calculus with labeled arguments and multi-abstractions which makes it comfortable to use the scripting API [3]. Having designed our own language also allowed us to integrate a few domain-specific extensions, to display helpful error messages and to generate a browsable documentation of the scripting API. In practice, many of the users of Liquidsoap are motivated by creating a radio and not very familiar with programming, so it can be considered that the design of the language was a success from this point of view.

An other motivation was to ensure some safety properties of the stream generation. A script in Liquidsoap describes a system that is intended to run for months, some parts of whose rarely triggered, and it would be very disappointing to notice a typo or a basic type error only after a particular part of the code is used for an important event. In order to ensure essential safety properties, the language is statically and strongly typed. We want to put as much static analysis as possible, as long as it doesn't put the burden on the user, *i.e.*, all types should be inferred. As we shall see, Liquidsoap also provides a few useful dynamic analysis.

The current paper can be read without a prior understanding of the language and its typing system, a detailed presentation can however be found in [3]. A basic knowledge of programming languages should be enough to understand the few examples presented in this paper, which construct sources using built-in operators of our language. For example, the following script defines two elementary sources, respectively reading from an HTTP stream and a playlist of files, composed in a fallback and filtered through a volume normalizer. The resulting

stream is sent to an Icecast server which broadcasts the stream to listeners, and saved in a local backup file:

```
s = normalize(fallback([input.http("http://other.net/radio"),
                        playlist("listing.txt")]))
output.icecast(%vorbis,mount="myradio",s)
output.file(%vorbis,"backup.mp3",s)
```

The graph underlying the system resulting from the execution of that script is shown in Figure 1. Note that the two output functions build a new source: these sources generate the same stream as `s`, but are active and have the side effect of encoding and sending the stream, respectively to an Icecast server and a file. A few remarks on the syntax: the notation `[...]` denotes a list, `mount` is a label (the name of an argument of the function `output.icecast`) and `%vorbis` is an encoding format parameter whose meaning is explained in Section 2 (recall that Vorbis is a compressed format for audio, similar to MP3).

1.3 Functional transitions

Liquidsoap is a *functional* programming language and a particularly interesting application of this is the case of *transitions*. Instead of simply sequencing tracks, one may want a smoother transition. For example, a *crossfade* consists in mixing the end of the old source, whose volume is faded out, with the beginning of the new one, whose volume is faded up (see Figure 2). But there is a wide variety of other possible transitions: a delay might be added, jingles may be inserted, etc.

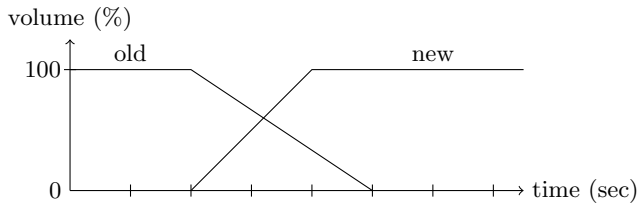

Fig. 2. A crossfade transition between two tracks

A solution that is both simple and flexible is to allow the user to specify a transition as a function that combines two sources representing the old and new tracks. We illustrate this feature with an example involving transitions used when switching from one source to another in a `fallback`. This is particularly useful when the fallback is track insensitive, *i.e.*, performs switching as soon as possible, without waiting for the end of a track. The following code defines a fallback source which performs a crossfade when switching from one source to another:

```
def crossfade(old,new) =
  add([fade.initial(duration=2.,new),fade.final(duration=3.,old)])
```

```
end
t = [crossfade,crossfade]
f = fallback(track_sensitive=false,transitions=t,[r,s])
```

Because any function can be used to define a transition, the possibilities are numerous. A striking example from the standard library of Liquidsoap scripts is the operator `smooth_add`, which takes as argument a main source (*e.g.*, musical tracks) and a special interruption source (*e.g.*, news items). When a new interruption is available, `smooth_add` gradually reduces the volume of the main source to put it in the background, and superposes the interruption. The reverse is performed at the end of the interruption. This very appreciated effect is programmed using the same building blocks as in the previous example.

1.4 Efficient implementation

An important aspect of the implementation is efficiency concerning both CPU and memory usage. The streams manipulated can have high data rates (a typical video stream needs 30Mo/s) and avoiding multiple copies of stream data is crucial.

In Liquidsoap, streams are computed using *frames*, which are data buffers representing a portion of stream portion of fixed duration. Abstractly, sources produce a stream by producing a sequence of frames. However, in the implementation a source is *passed* a frame that it has to fill. Thus, we avoid unnecessary copies and memory allocations. Active sources, which are the initiators of streaming, initially allocate one frame, and keep re-using it to get stream data from their input source. Then, most sources do not need to allocate their own frame, they simply pass frames along and modify their content in place. However, this simple mechanism does not work when a source is *shared*, *i.e.*, it is the input source of several sources. This is the case of the `normalize` node in the graph of Figure 1 (which happens to be shared by active sources). In that case, we use a *caching* mechanism: the source will have its own *cache* frame for storing its current output. The first time that the source is asked to fill a frame, it fills its internal cache and copies the data from it; in subsequent calls it simply fills the frame with the data computed during the first call. Once all the filling operations have been done, the sources are informed that the stream has moved on to the next frame and can forget their cache.

With this system, frames are created once for all, one for each active source plus one for each source that is shared and should thus perform caching — of course, some sources might also need another frame depending on their behavior. Sharing is detected automatically when the source is initialized for streaming. We do not detail this analysis, but note that the dynamic reconfigurations of the streaming system (notably caused by transitions) make it non-trivial to anticipate all possible sharing situations without over-approximating too much.

2 Heterogeneous stream contents

In Liquidsoap, streams can contain data of various nature. The typical example is the case of video streams which usually contain both images and audio samples. We also support MIDI streams (which contain musical notes) and it would be easy to add other kinds of content. It is desirable to allow sources of different content kinds within a streaming system, which makes it necessary to introduce a typing discipline in order to ensure the consistency of stream contents across sources.

The nature of data in streams is described by its *content type*, which is a triple of natural numbers indicating the number of audio, video and midi channels. A stream may not always contain data of the same type. For instance, the `playlist` operator might rely on decoding files of heterogeneous content, *e.g.*, mono and stereo audio files. In order to specify how content types are allowed to change over time in a stream, we use *arities*, which are essentially natural numbers extended with a special symbol $\star$:

$$a \quad ::= \quad \star \mid 0 \mid S(a)$$

An arity is *variable* if it contains $\star$, otherwise it is an usual natural number, and is *fixed*. A *content kind* is a triple of arities, and specifies which content types are acceptable. For example, $(S(S(0)), S(\star), \star)$ is the content kind meaning "2 audio channels, at least one video channel and any number of MIDI channels". This is formalized through the subtyping relation defined in Figure 3: $T <: K$ means that the content kind T is allowed by K. More generally, $K <: K'$ expresses that K is more permissive than K', which implies that a source of content kind K can safely be seen as one of content kind K'.

$$\frac{}{0 <: 0} \qquad \frac{A <: A'}{S(A) <: S(A')} \qquad \frac{}{\star <: \star} \qquad \frac{}{0 <: \star} \qquad \frac{A <: \star}{S(A) <: \star}$$

$$\frac{A <: A' \quad B <: B' \quad C <: C'}{(A, B, C) <: (A', B', C')}$$

Fig. 3. Subtyping relation on arities

When created, sources are given their expected content kind. Of course, some assignments are invalid. For example, a pure audio source cannot accept a content kind which requires video channels, and many operators cannot produce a stream of an other kind than that of their input source. Also, some sources have to operate on input streams that have a fixed kind – a kind is said to be fixed when all of its components are. This is the case of the `echo` operator which produces echo on sound and has a internal buffer of a fixed format for storing past sound, or sound card inputs/outputs which have to initialize the sound card for a specific number of channels. Also note that passing the expected content kind is important because some sources behave differently depending on their kind, as shown with the previous example.

Integration in the language. To ensure that streaming systems built from user scripts will never encounter situations where a source receives data that it cannot handle, we leverage various features of our type system. By doing so, we guarantee statically that content type mismatches never happen. The content kinds are reflected into types, and used as parameters of the `source` type. In order to express the types of our various operators, we use a couple features of type systems (see [5] for extensive details). As expected, the above subtyping relation is integrated into the subtyping on arbitrary Liquidsoap types. We illustrate various content kinds in the examples of Figure 4:

- The operator `swap` exchanges the two channels of a stereo audio stream. Its type is quite straightforward: it operates on streams with exactly two audio channels.
- Liquidsoap supports polymorphism *à la* ML. We use it in combination with constraints to allow arbitrary arities. The notation `'*a` stands for a universal variable (denoted by `'a`) to which a type constraint is attached, expressing that it should only be instantiated with arities. For example, the operator `on_metadata` does not rely at all on the content of the stream, since it is simply in charge of calling a handler on each of its metadata packets – in the figure, `handler` is a shortcut for `([string*string]) -> unit`.
- When an operator, such as `echo`, requires a fixed content type, we use another type constraint. The resulting constrained universal variable is denoted by `'#a` and can only be instantiated with fixed arities.
- The case of the `greyscale` operator, which converts a color video into greyscale, shows how we can require at least one video channel in types. Here, `'*b+1` is simply a notation for `S('*b)`.
- Finally, the case of `output.file` (as well as several other outputs which encode their data before sending it to various media) is quite interesting. Here, the expected content kind depends on the format the stream is being encoded to, which is given as first argument of the operator. Since typing the functions generating formats would require dependent types (the number of channels would be given as argument) and break type inference, we have introduced particular constants for type formats with syntactic sugar for them to appear like functions – similar ideas are for example used to type the `printf` function in OCaml. For example, `output.file(%vorbis,"stereo.ogg",s)` requires that `s` has type `source(2,0,0)` because `%vorbis` alone has type `format(2,0,0)`, but `output.file(%vorbis(channels=1),"mono.ogg",s)` requires that there is only one audio channel; we also have video formats such as `%theora`.

```
       swap : (source(2,0,0)) -> source(2,0,0)
on_metadata : (handler,source('*a,'*b,'*c)) -> source('*a,'*b,'*c)
       echo : (delay:float,source('#a,0,0)) -> source('#a,0,0)
  greyscale : (source('*a,'*b+1,'*c)) -> source('*a,'*b+1,'*c)
output.file : (format('*a,'*b,'*c),string,source('*a,'*b,'*c))->
              source('*a,'*b,'*c)
```

Fig. 4. Types for some operators

These advanced features of the type system are statically inferred, which means that the gain in safety does not add any burden on users. As said above, content kinds have an influence on the behavior of sources — polymorphism is said to be *non-parametric*. In practice, this means that static types must be maintained throughout the execution of a script. This rather unusual aspect serves us as an overloading mechanism: the only way to remove content kinds from execution would be to duplicate our current collection of operators with a different one for each possible type instantiation.

Ideally, we would like to add some more properties to be statically checked by typing. But it is sometimes difficult to enrich the type system while keeping a natural syntax and the ability to infer types. For example, Liquidsoap checks that active sources are *infallible*, *i.e.*, always have data available in their input stream, and this check is currently done by a flow analysis on instantiated sources and not typing. Another example is clocks which are described next section.

3 Clocks

Up to now, we have only described streaming systems where there is a unique global clock. In such systems, time flows at the same rate for all sources. By default, this rate corresponds to the wallclock time, which is appropriate for a live broadcast, but it does not need to be so. For example, when producing a file from other files, one might want the time rate to be as fast as the CPU allows.

While having a global clock suffices in many situations, there are a couple of reasons why a streaming system might involve multiple clocks or time flows. The first reason is external to liquidsoap: there is simply not a unique notion of time in the real world. A computer's internal clock indicates a slightly different time than your watch or another computer's clock. Moreover, when communicating with a remote computer, network latency causes a perceived time distortion. Even within a single computer there are several clocks: notably, each soundcard has its own clock, which will tick at a slightly different rate than the main clock of the computer. Since liquidsoap communicates with soundcards and remote computers, it has to take those mismatches into account.

There are also some reasons that are purely internal to liquidsoap: in order to produce a stream at a given speed, a source might need to obtain data from another source at a different rate. This is obvious for an operator that speeds up or slows down audio, but is also needed in more subtle cases such as a crossfading operator. A variant of the operator described in Section 1.3 might combine a portion of the end of a track with the beginning of the next track *of the same source* to create a transition between tracks. During the lapse of time where the operator combines data from an end of track with the beginning of the other other, the crossing operator needs to read both the stream data of the current track and the data of the next track, thus reading twice as much stream data as in normal time. After ten tracks, with a crossing duration of six seconds, one

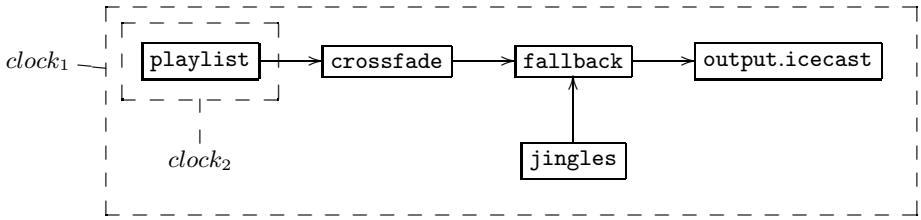

Fig. 5. A streaming system with two clocks

more minute will have passed for the source compared to the time of the crossing operator.

3.1 Model

In order to avoid inconsistencies caused by time differences, while maintaining a simple and efficient execution model for its sources, liquidsoap works under the restriction that one source belongs to a unique clock, fixed once for all when the source is created. Sources from different clocks cannot communicate using the normal streaming protocol, since it is organized around clock cycles: each clock is responsible for animating its own active sources and has full control on how it does it.

In the graphical representation of streaming systems, clocks induce a partition of sources represented by a notion of locality or box, and clock dependencies are represented by nesting. For example, the graph shown in Figure 5 corresponds to the stream generators built by the following script:

```
output.icecast(%vorbis,mount="myradio",
  fallback([crossfade(playlist("some.txt")),jingles]))
```

There, $clock_2$ was created specifically for the crossfading operator; the rate of that clock is controlled by that operator, which can hence accelerate it around track changes without any risk of inconsistency. $clock_1$ is simply a wallclock, so that the main stream is produced following the real time rate.

A clock is *active* if it ticks by itself, therefore running its sources constantly; this is the case of wallclocks or soundcard clocks. We say that a clock depends on another one if its animation (and thus time rate) depends on it. Active sources do not depend on other sources, and dependencies must be acyclic. In the above example, the ticking of $clock_2$ is provoked by that of $clock_1$, and freezes when the fallback is playing jingles. Although nothing forces it in the model, it makes more sense if each passive source depends (possibly indirectly) on an active one, and all sources without dependencies are active. Those assumptions are in fact guaranteed to hold for the systems built using the Liquidsoap language.

From an implementation viewpoint, each active clock launches its own streaming thread. Hence, clocks provide a way to split the generation of one or several streams across several threads, and hence multiple CPU cores.This powerful possibility is made available to the user through the intuitive notion of clock. As we

shall see in the next section, the script writer never needs to specify clocks unless he explicitly wants a particular setup, and Liquidsoap automatically checks that clock assignements are correct.

3.2 Clock assignment

Clocks are not represented in the type of Liquidsoap sources. Although it would be nice to statically check clock assignment, type inference would not be possible without technical annotations from the user. Instead, clocks are assigned upon source creation. Some sources require to belong to a particular, definite clock, such as the wallclock, or the clock corresponding to a sound card. Most sources simply require that their clock is the same as their input sources. Since clocks often cannot be inferred bottom-up, we use a notion of clock variable that can be left undefined. Clock variables reflect the required clock dependencies, which are maintained during the inference process.

Two errors can occur during this phase. Although they are runtime errors that could be raised in the middle of streaming when new sources are created (*e.g.*, by means of a transition), this usually only happens during the initial construction. The first error is raised when two different known clocks need to be unified. For example, in the following script, the ALSA input is required to belong to the ALSA clock and `crossfade`'s internal clock at the same time:

```
output.file(%vorbis,"record.ogg",crossfade(input.alsa()))
```

The other possible error happens when unifying two unknown clock variables if one depends on the other – in unification terminology, this is an *occurs-check* failure. A simple example of that situation is the script `add([s,crossfade(s)])` where the two mixed sources respectively have clocks c and X_c where c is the clock created for the crossfading operator and X_c is the variable representing the clock to which the crossfading belongs, on which c depends.

After this inference phase, it is possible that some clocks are still unknown. Remaining variables are thus forcibly assigned to the default wallclock, before that all new sources are prepared for streaming by their respective clocks.

4 Related work

Liquidsoap is obviously different from classical tools such as Ices or Darkice in that it offers the user the freedom to assemble a stream for a variety of operators, through a scripting language rather than traditional configuration files.

Liquidsoap has more similarities with multimedia streaming libraries and digital signal processing (DSP) languages. The GStreamer library [6] defines a model of stream, and its API can be used to define streaming systems in various programming languages (primarily coded in C, the library has also been ported to many other languages). Faust [4] provides a high-level functional programming language for describing stream processing devices, and compiles this language down to C++, which enables an integration with various other systems. It is also

worth mentioning Chuck [7], a DSP programming language with an emphasis on live coding (dynamic code update). Besides a different approach and target application, Liquidsoap differs more deeply from these tools. The notion of source provides a richer way of generating streams, providing and relying on the additional notions of tracks and metadata; also recall the ability to momentarily stop streaming, and the possibility to dynamically create or destroy sources. It would be very interesting to interface Liquidsoap with the above mentionned tools, or import some of their techniques. This could certainly be done for simple operators such as DSP, and would allow us to program them efficiently and declaratively from the scripting language rather than in OCaml.

5 Conclusion

We have presented the main ideas behind the design of Liquidsoap, a tool used by many netradios worldwide as well as in some academic work [1]. We believe that Liquidsoap demonstrates the potential of building applications as domain-specific languages. It also shows that very rich type systems can be put to work usefully even in tools not designed for programmers: although most Liquidsoap users have a limited understanding of our type system, they are able to fix their mistakes when an error is reported — errors might be difficult to read but they have the merit of signaling real problems.

Of course, there are many other reasons behind the success of Liquidsoap, including a wide variety of features plugged onto the basic organization described here. Some of the future work on Liquidsoap lies there: integration with other tools, graphical interfaces, documentation, etc. But we are also planning some improvements of the language. One of the goals is to make it possible to express more operators directly in Liquidsoap instead of OCaml, bringing more customizability to the users. Also, Liquidsoap offers a server through which many sources offer various services. An interesting way to structure more this very useful system would be to consider sources as objects whose methods are services, and type them accordingly.

References

1. C. Baccigalupo and E. Plaza. A case-based song scheduler for group customised radio. *Case-Based Reasoning Research and Development*, pages 433–448, 2007.
2. D. Baelde, R. Beauxis, S. Mimram, and al. Liquidsoap. `http://savonet.sf.net/`.
3. D. Baelde and S. Mimram. De la webradio lambda à la λ-webradio. In *Journées Francophones des Langages Applicatifs (JFLA'08)*, pages 47–62, 2008.
4. Y. Orlarey, D. Fober, and S. Letz. FAUST: an Efficient Functional Approach to DSP Programming. *New Computational Paradigms for Computer Music*, 2009.
5. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
6. W. Taymans, S. Baker, A. Wingo, R. Bultje, and S. Kost. GStreamer application development manual.
7. G. Wang, P. Cook, et al. ChucK: A concurrent, on-the-fly audio programming language. In *Proc. of Int. Comp. Music Conf.*, pages 219–226, 2003.